File Handling

File

- File is a named location on disk to store related information
- It is used to permanently store data in non-volatile memory
- In Python, a file operation take place in the following order

    Open a file

    Read or write (perform operation)

    Close the file

Open a file

- We have to open a file using built-in function **open()**
- This function returns a file object, also called a handle, it is used to read or modify the file
- We can specify the mode while opening a file.

  r – Read

  w – Write

  a – Append

Syntax:

```
file object= open(filename [,accessmode][, buffering])
```

filename: name of the file that we want to access

accessmode: read, write, append etc.

buffering: 0 – no buffering

1- line buffering

integer greater than 1- buffering action is performed with the indicated buffer size

Example:

```
f=open("abc.txt",'r') # open file in current directory
f=open("C:/Python33/sample.txt",'r')#specifying full path
```

Attributes of file object

Once a file is opened , we have one file object, which contain various info related to that file

1. file.closed – Returns true if the file is closed, False otherwise
2. file.mode – Returns access mode with wich file was opened
3. file.name- Name of the file
4. file.softspace – Returns 0 if space is explicitly required with print,
   1 otherwise

Attributes of file object

Example

```
fo.open("abc.txt","w")
print("Name of the file:",fo.name)
print("Closed or not :",fo.closed)
print("Opening mode :",fo.mode)
print("Softspace flag :",fo.softspace)
```

Name of the file: abc.txt
Closed or not:False
Opening mode:w
Softspace flag:0

Modes for opening a file

Mode	Description
r	Reading onl
rb	Reading only in binary format
r+	Reading and writing
rb+	Reading and writing in binary format
w	Writing only
wb	Writing only in binary Overwrites the file if exists. If the file does not exist, create a new file for writing.

Modes for opening a file

Mode	Description
w+	both writing and reading. Overwrites the file if exists. If the file does not exist, create a new file for reading and the file writing
wb+	Writing and reading in binary format
a	Open a file for appending
ab	Appending in binary format
a+	Appending and reading
ab+	Appending and reading in binary format
x	Exclusive creation

Closing a File

- File closing is done with close() method

Syntax :

```
fileobject.close()
```

Example:

```
f=open("bin.tx",wb)
proint("Name of the file:",fo.name)
fo.close()
print("File closed")
```

Output

```
Name of the file: bin.txt
File closed
```

Reading File

- File object includes the following methods to read data from the file.
- **read(chars):** reads the specified number of characters starting from the current position.
- **readline():** reads the characters starting from the current reading position up to a newline character.
- **readlines():** reads all lines until the end of file and returns a list object.

File reading in Python

```
# read the entire file as one string
with open('filename.txt') as f:
data = f.read()
# Iterate over the lines of the File
with open('filename.txt') as f:
for line in f :
     print(line, end=' ‘)
# process the lines
```

- The following example demonstrates reading a line from the file

```
>>>f=open('C:\myfile.txt') # opening a file
>>> line1 = f.readline() # reading a line
>>> line1
'This is the first line. \n'
>>> line2 = f.readline() # reading a line
>>> line2
'This is the second line.\n'
>>> line3 = f.readline() # reading a line
>>> line3
'This is the third line.'
>>> line4 = f.readline() # reading a line
>>> line4
''
>>> f.close() # closing file object
```

we have to open the file in 'r' mode. The readline() method will return the first line, and then will point to the second line.

Reading all Lines

- The following reads all lines using the readlines() function

```
f= open('C:\myfile.txt') # opening a file
>>> lines = f.readlines() # reading all lines
>>> lines
'This is the first line. \nThis is the second
line.\nThis is the third line.'
>>> f.close() # closing file object
```

Use for loop to read a file

```
f=open('C:\myfile.txt')
for line in f:
        print(line)
f.close()
```

File writing in Python

- Similarly, for writing data to files, we have to use open() with 'wt' mode
- Also, we have to use the write() function to write into a file.

```
# Write text data to a file
with open('filename.txt' , 'wt') as f:
        f.write ('hi there, this is a first line of file.\n’)
        f.write ('and another line.\n’)
```

```
Output
hi there, this is a first line of file.
and another line.
```

with statement

- With statement is used when we have two related operations which we would like to execute as a pair, with a block of code in between

Example: opening a file, manipulating a file and closing it

```
with open("output.txt","w") as f:
        f.write("Hello Python!")
```

- The above statement automatically close the file after the nested block of code.
- The advantage of using with statement is that it is guaranteed to close the file.
- If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler

Writing a file that does not exist

- The problem can be easily solved by using another mode - technique, i.e., the 'x' mode to open a file instead of 'w' mode.

```
with open('filename' , 'wt') as f:
        f.write ('Hello, This is sample content.\n’)
 # This will create an error that the file 'filename' doesn't exist.


with open ('filename.txt' , 'xt') as f:
        f.write ('Hello, This is sample content.\n’)
```

In binary mode, we should use '**xb**' instead of '**xt**'.

Renaming a file

- The **os** module Python provides methods that help to perform file-processing operations, such as renaming and deleting.
- To rename an existing file, rename() method is used
- It takes two arguments, current file name and new file name

Syntax:

```
os.rename(current_file_name, new_file_name)
```

Example:

```
import os
os.rename("test.txt","Newtest.txt")
print("File renamed")
```

Deleting a file

- We can delete a file by using remove() method
- Syntax:

```
os.remove(filename)
```

Example:

```
import os
os.remove("Newtest.txt")
print("File deleted")
```

- Write a Python program to reverse a string. Import the module to reverse a string input by the user
- Write a program that asks the user to enter a list of integers. Do the following:
- (a) Print the total number of items in the list.
- (b) Print the last item in the list.
- (c) Print the list in reverse order.
- (d) Print Yes if the list contains a 5 and No otherwise.
- (e) Print the number of fives in the list.
- (f) Remove the first and last items from the list, sort the remaining items, andprint the result.

- Write a Python program to remove elements from set using remove(). discard() and pop()
- Write a program to return a new set of identical items from two sets
- Write a program to perform set operations; union, intersection, difference, symmetric difference
- Python program to delete an element from a list by index
- Python program to check whether a string is palindrome or not
- Python program to implement matrix addition
- Python program to implement matrix multiplication
- Write a Python program to check if a set is a subset of another set
- Write a Python program to use of frozensets.
- Write a Python program to find maximum and the minimum value in a set
- Write a Python program to find the index of an item of a tuple

Reading and writing binary File

```
#For Read Only Mode
 with open('filename with Path', mode='rb’)
#For Read-Write Mode
with open('Filename with path', mode=‘wb+’)
```

- ab+ - Appending and reading in binary format
- ab - Appending in binary format
- wb – Writing only in binary format

Read Binary File

```
#Opening a binary File in Read Only Mode
with open('C:\Test\\slick.bin', mode='rb') as binaryFile:
    lines = binaryFile.readlines()
    print(lines)
```

We are reading all the lines in the file in the lines variable above and print all the lines read in form of a list.

Write Binary File

```
#Opening a binary File in Write Mode
with open('C:\Test\\TestBinaryFile.bin', mode='wb') as binaryFile:
    #Assiging a Binary String
    lineToWrite = b'You are on Coduber.'
    #Writing the binary String to File.
    binaryFile.write(lineToWrite)
    #Closing the File after Writing
    binaryFile.close()
```

Pickle Module

- You may sometimes need to send complex object hierarchies over a network or save the internal state of your objects to a disk or database for later use.
- To accomplish this, you can use serialization,
- It is fully supported by the standard library Python pickle module.

- The **serialization (marshalling)** process is a way to convert a data structure into a linear form that can be stored or transmitted over a network.
- In Python, serialization allows you to take a complex object structure and transform it into a stream of bytes that can be saved to a disk or sent over a network.
- The reverse process, which takes a stream of bytes and converts it back into a data structure, is called **deserialization** or **unmarshalling**.

Pickle Module

- It is a way to serialize and deserialize Python objects.
- It serializes the Python object in a binary format, due to which it is not human-readable.
- It is faster and it also works with custom-defined objects.
- The Python pickle module is a better choice for serialization and deserialization of python objects.
- If you don’t need a human-readable format or if you need to serialize custom objects then it is recommended to use the pickle module.

- The Python pickle module basically consists of four methods:

1. pickle.dump(obj, file, protocol=None, *, fix_imports=True, buffer_callback=None)

2. pickle.dumps(obj, protocol=None, *, fix_imports=True,buffer_callback=None)

3. pickle.load(file, *, fix_imports=True, encoding="ASCII",errors="strict", buffers=None)

4. pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII",errors="strict", buffers=None)

- The first two methods are used during the pickling process, and the other two are used during unpickling.
- The only difference between **dump() and dumps()** is:
- The first creates a file containing the serialization result, whereas the second returns a string.
- The same concept also applies to **load() and loads():**
- The first one reads a file to start the unpickling process, and the second one operates on a string.

- The example below shows how you can instantiate the class and pickle the instance to get a plain string.
- After pickling the class, you can change the value of its attributes without affecting the pickled string.
- You can then unpickle the pickled string in another variable, restoring an exact copy of the previously pickled class:

```
# pickling.py
import pickle
class example_class:
    a_number = 35
    a_string = "hey"
    a_list = [1, 2, 3]
    a_dict = {"first": "a", "second": 2, "third": [1, 2, 3]}
    a_tuple = (22, 23)
my_object = example_class()
my_pickled_object = pickle.dumps(my_object)  # Pickling the object
print(f"This is my pickled object:\n{my_pickled_object}\n")
my_object.a_dict = None
my_unpickled_object = pickle.loads(my_pickled_object)  # Unpickling the object
print(f"This is a_dict of the unpickled object:\n{my_unpickled_object.a_dict} n\")
```

Reading and writing csv file in Python

- CSV (stands for comma separated values) format is a commonly used data format used by spreadsheets.
- The csv module in Python’s standard library presents classes and methods to perform read/write operations on CSV files.

writer()

- This function in csv module returns a writer object that converts data into a string and stores in a file object.
- The function needs a file object with write permission as a parameter.
- Every row written in the file issues a newline character.
- To prevent additional space between lines, newline parameter is set to ‘ ’.

writerow()

- This function writes items in an iterable (list, tuple or string) ,separating them by comma character.

writerows()

- This function takes a list of iterables as parameter and writes each item as a comma separated line of items in the file.
- First a file is opened in ‘w’ mode.
- This file is used to obtain writer object.
- Each tuple in list of tuples is then written to file using writerow() method.

Example

```
import csv
persons=[('Lata',22,45),('Anil',21,56),('John',20,60)]
csvfile=open('persons.csv','w', newline='')
obj=csv.writer(csvfile)
for person in persons:
    obj.writerow(person)
csvfile.close()
```

- This will create ‘persons.csv’ file in current directory. It will show following data.

Output

Lata,22,45

Anil,21,56

John,20,60

read()

- This function returns a reader object which returns an iterator of lines in the csv file.
- Using the regular for loop, all lines in the file are displayed in following example

```
csvfile=open('persons.csv','r', newline='')
obj=csv.reader(csvfile)
for row in obj:
      print (row)
```

output

['Lata', '22', '45']

['Anil', '21', '56']

['John', '20', '60']

DictWriter()

- This function returns a DictWriter object.
- It is similar to writer object, but the rows are mapped to dictionary object.
- The function needs a file object with write permission and a list of keys used in dictionary as fieldnames parameter.
- This is used to write first line in the file as header.

writeheader()

- This method writes list of keys in dictionary as a comma separated line as first line in the file.

DictReader()

- This function returns a DictReader object from the underlying CSV file.
- As in case of reader object, this one is also an iterator, using which contents of the file are retrieved.

```
>>> csvfile = open('persons.csv','r', newline='')
>>> obj = csv.DictReader(csvfile)
```

- The class provides fieldnames attribute, returning the dictionary keys used as header of file.

```
>>> obj.fieldnames
```

Output

['name', 'age', 'marks']

os.path module

- The OS module in Python provides functions for interacting with the operating system.
- OS comes under Python's standard utility modules.
- This module provides a portable way of using operating system-dependent functionality.
- The '**os' and 'os.path'** modules include many functions to interact with the file system.

os.path module

- The os.path module is a very extensively used module that is handy when processing files from different places in the system.
- It is used for different purposes such as for merging, normalizing and retrieving path names in python .
- All of these functions accept either only bytes or only string objects as their parameters.
- Its results are specific to the OS on which it is being run.

os.path.basename

- This function gives us the last part of the path which may be a folder or a file name.

Example

```
import os

# In windows

file=os.path.basename("C:\\Users\\xyz\\Documents\\MyWeb
Sites\\intro.html")

print(file)

# In nix*

file = os.path.basename("/Documents/MyWebSites/music.txt")

print(file)
```

Output

My Web Sites

intro.html

MyWebSites

music.txt

os.path.dirname

- This function gives us the directory name where the folder or file is located.

Example

```
import os
# In windows
DIR = os.path.dirname("C:\\Users\\xyz\\Documents\\My Web Sites")
print(DIR)
# In nix*
DIR = os.path.dirname("/Documents/MyWebSites")
print(DIR)
```

Output

C:\Users\xyz\Documents

/Documents

- **Other functions**
- os.path.abspath(path)
- os.path.commonpath(paths)
- os.path.commonprefix(list)
- os.path.exists(path)
- os.path.lexists(path)
- os.path.expanduser(path)
- os.path.expanduser(path)
- os.path.isdir(path) ……etc